King Mongkut's University of Technology Thonburi

Tree Data Structures

CSC 209 Data Structures

Dr. Kulwadee Somboonviwat

School of Information Technology, KMUTT

kulwadee.som [at] sit.kmutt.ac.th

Lecture Plan

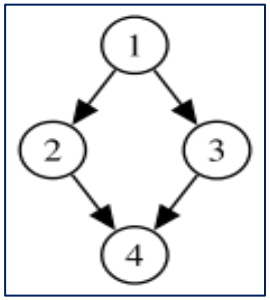
- Tree Data Structure
- Binary Search Tree
- Querying a Binary Search Tree
- Insertion and Deletion
- Summary

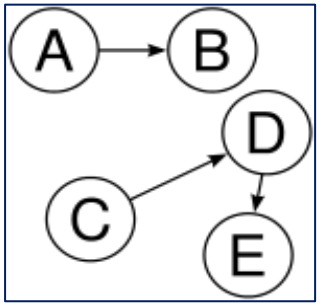
From linear to non-linear structures

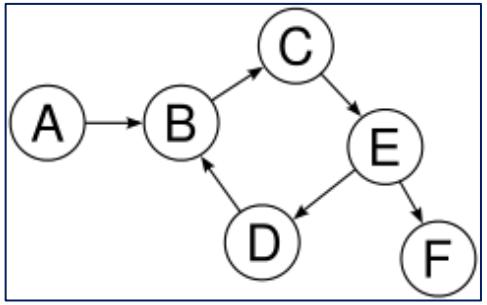
- **Array** – a *static* data structure that can be accessed randomly and easy to implement
- **Linked List** – a *dynamic* data structure that is ideal for applications requiring frequent *add, delete,* and *update* operations.
- A main disadvantage of using an array or linked list to manage data is *data access time*. Since both the arrays and linked lists are **linear structures**, the time needed to search for an element is proportional to the size of the data set.
- Trees – an extension of linked lists to allow hierarchical relationships among data items.

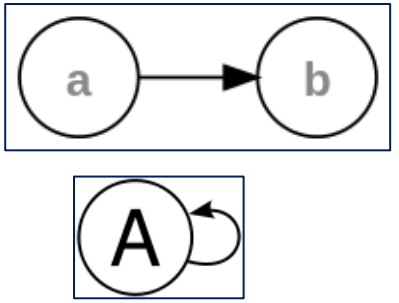
Trees

- A non-linear data structure made up of nodes or vertices and edges **without having any cycles**.
- The tree with no nodes is called **null** or **empty** tree.
- A tree that is not empty consist of a root node and potentially many levels of additional nodes that form a hierarchy.









Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)

Trees – Terminology

- **Root** – the top node in a tree.
- **Child** – a node directly connected to another node when moving away from the root.
- **Parent** – a node directly connected to another node when moving to the root.
- **Siblings** – a group of nodes with the same parent.
- **Descendant** – a node reachable by repeated proceeding from parent.
- **Ancestor** – a node reachable by repeated proceeding from child to parent.
- **Leaf** (or external node) – a node with no children.
- **Branch** (or internal node) – a node with at least one child.
- **Edge** – the connection between one node and another.
- **Path** – a sequence of nodes and edges connecting a node with a descendant.
- **Height of node** – the number of edges on the longest path between that node and a leaf
- **Height of tree** – the height of the root node of the tree.
- **Depth** – the number of edges from the tree's root to the node

Reference: https://en.wikipedia.org/wiki/Tree_(data_structure)

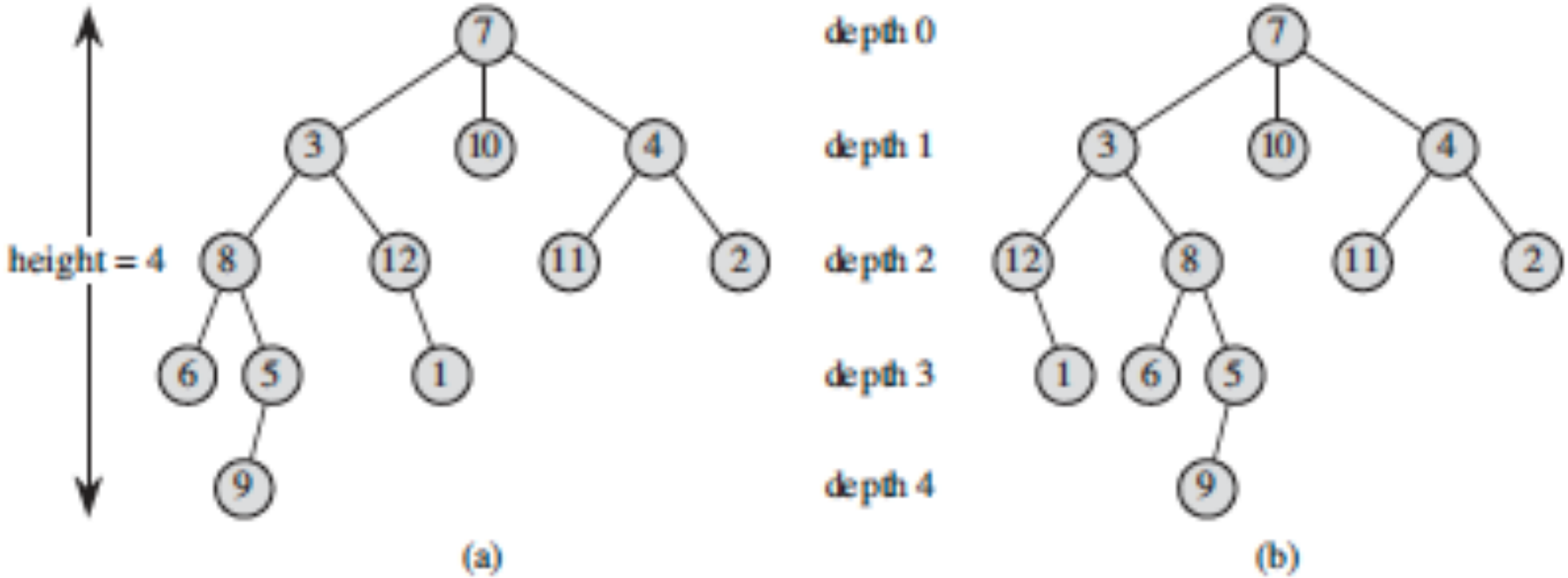
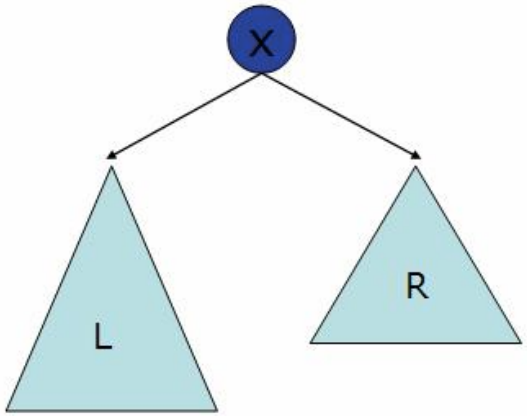


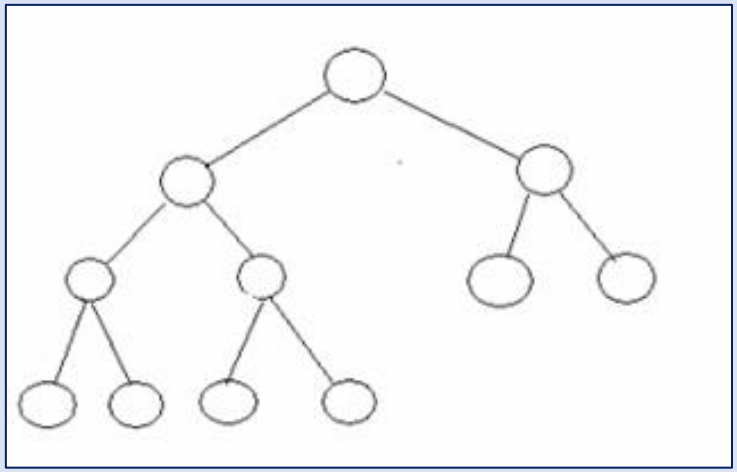
Figure B.6 Rooted and ordered trees. **(a)** A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right order of the children of a node matters; otherwise it doesn't. **(b)** Another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.

Binary Trees

- A tree where each node can have no more than *two* children.



A complete binary tree is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height h has between 2^h and $2^{(h+1)}$-1 nodes.



Reference: http://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson4_1.htm

Properties of Binary Trees

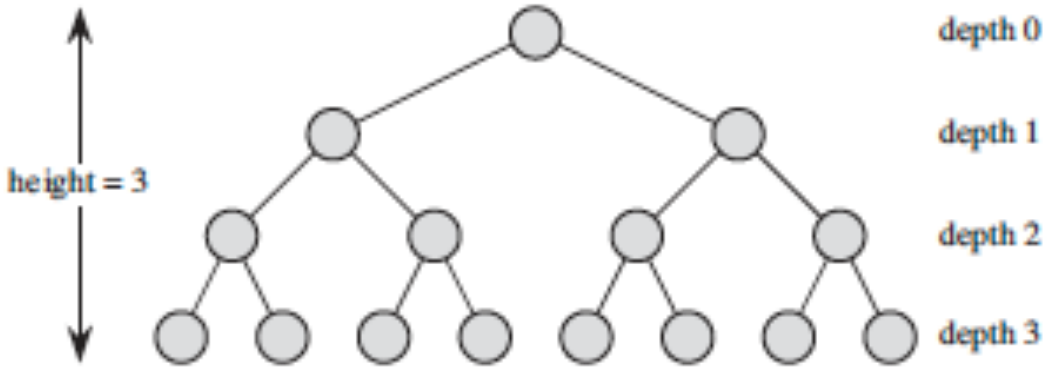


Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

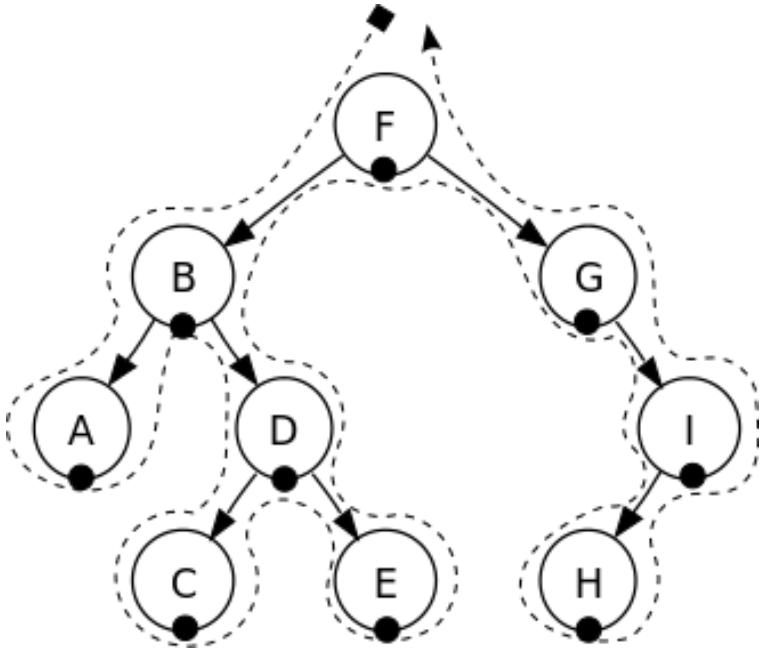
Let number of nodes in a binary tree = *N*

- Height of a complete binary tree with *N* nodes: $h = \lg (N+1) - 1$
- Number of internal node = $2^h - 1$

Tree Traversal: In-order

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

1. Check if the current node is empty / null.
2. Traverse the left subtree by recursively calling the in-order function.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order function.



In-order: A, B, C, D, E, F, G, H, I.

Reference: https://en.wikipedia.org/wiki/Tree_traversal

Applications of In-order Traversal

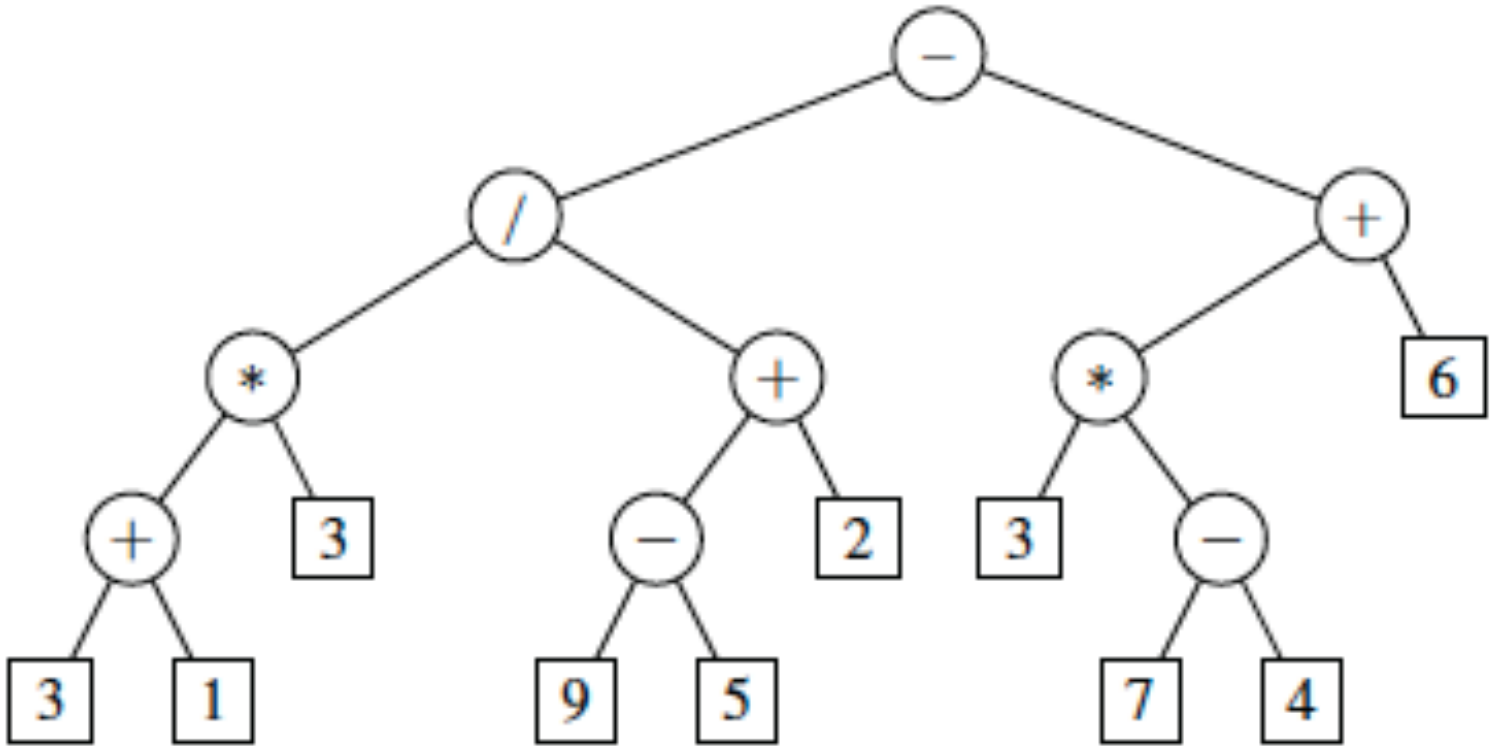
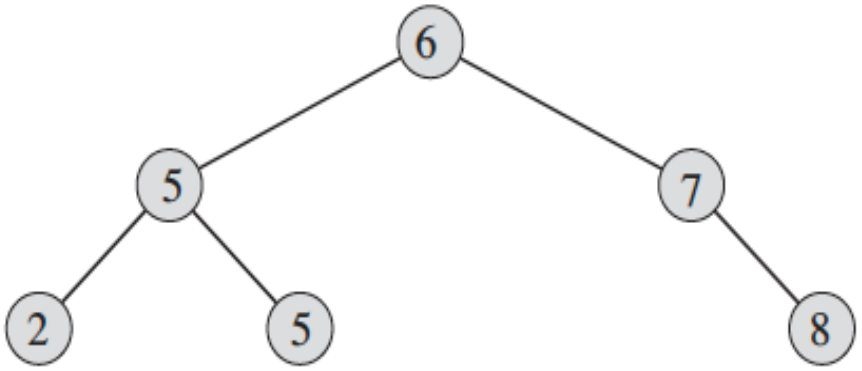


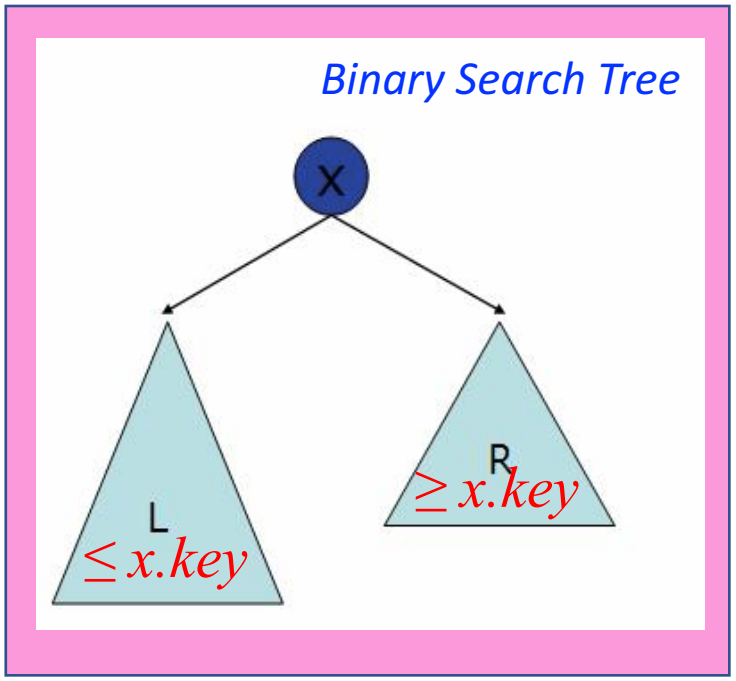
Figure 8.6: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))$. The value associated with the internal node labeled "/" is 2.

Applications of In-order Traversal

- In-order traversal over a Binary Search Trees (BST) retrieves data in sorted order

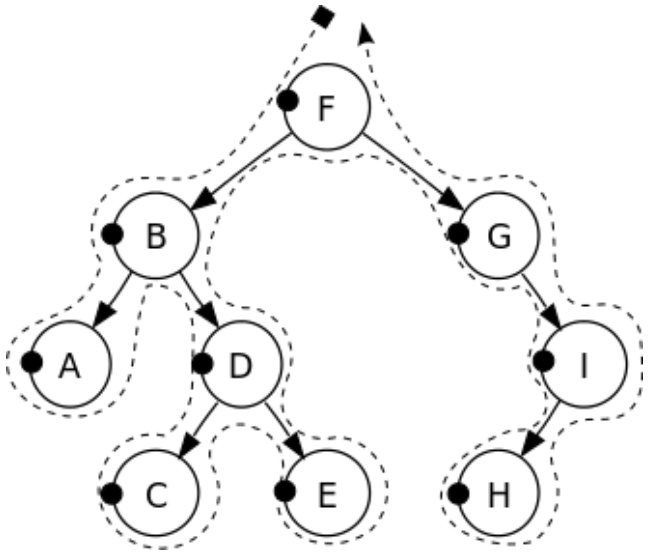


In-order: 2, 5, 5, 6, 7, 8



Tree Traversal: Pre-order

1. Check if the current node is empty / null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order function.
4. Traverse the right subtree by recursively calling the pre-order function.

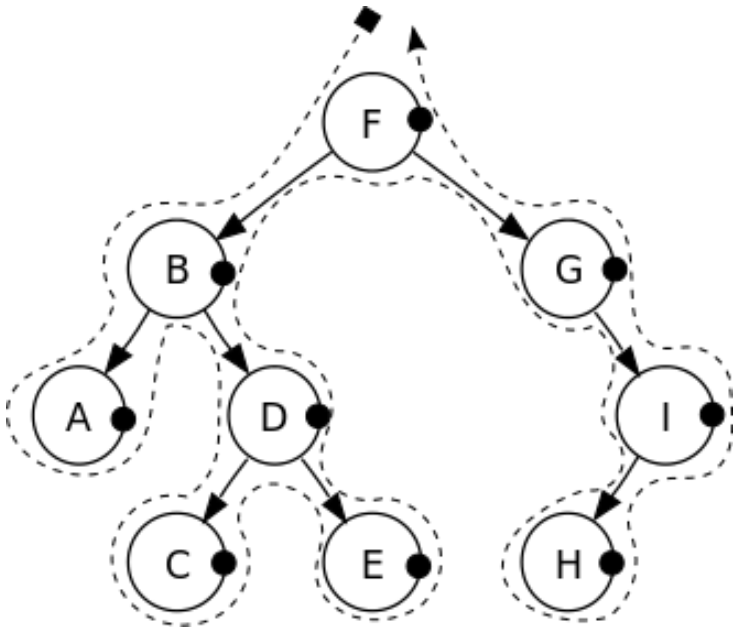


Pre-order: F, B, A, D, C, E, G, I, H.

Reference: https://en.wikipedia.org/wiki/Tree_traversal

Tree Traversal: Post-order

1. Check if the current node is empty / null.
2. Traverse the left subtree by recursively calling the post-order function.
3. Traverse the right subtree by recursively calling the post-order function.
4. Display the data part of the root (or current node).



Post-order: A, C, E, D, B, H, I, G, F.

Reference: https://en.wikipedia.org/wiki/Tree_traversal

Lecture Plan

- Tree Data Structure
- Binary Search Tree
- Querying a Binary Search Tree
- Insertion and Deletion
- Summary

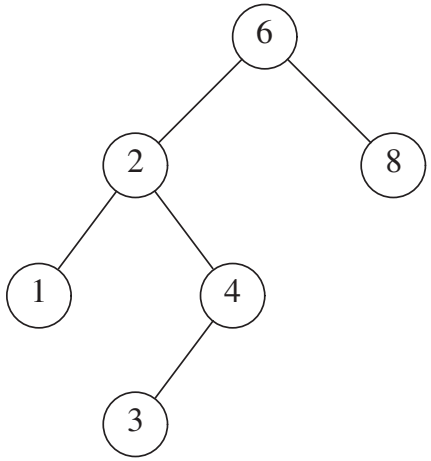
Searching

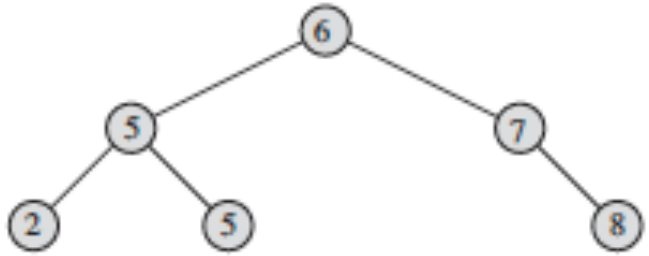
- Sequential search in array/linked list is too slow for large data
- Hashing is useful only for “exact” search operation.
- Heaps is useful only for “findMin” or “findMax” operations
- To support following operations, we need **search trees**.
  - Lookup-by-prefix
  - Find-in-range[key1, key2]
  - NearLookup(x): Returns the closest key to x when x is not in the database
  - Find the k-th smallest/largest key
  - Rank(x): How many keys are smaller than x?

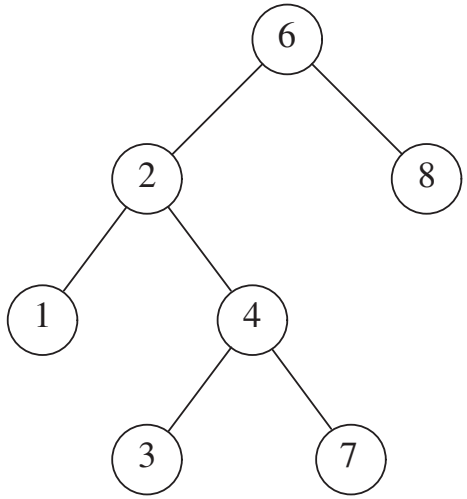
Binary Search Trees

- A binary tree that satisfies the **binary-search-tree property**
  - For any node *X*:
    - All keys in the left subtrees <= X.key
    - All keys in the right subtress >= X.key

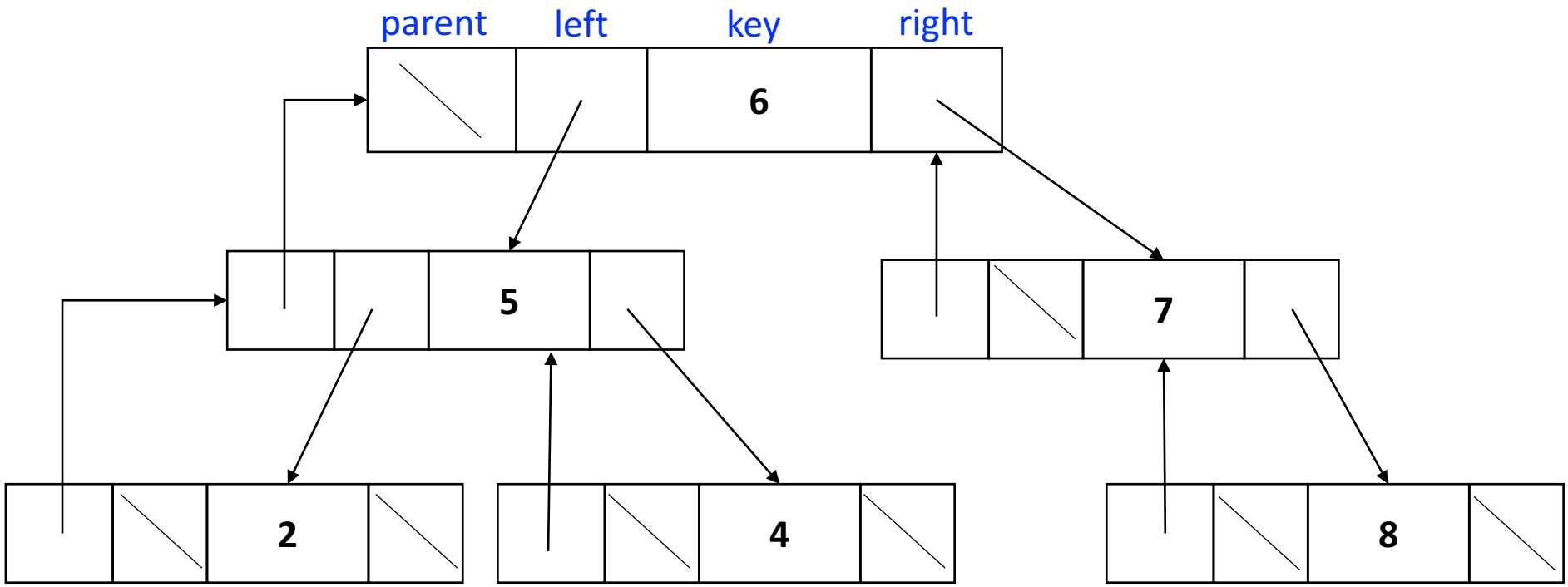
Exercise: which of the following are valid BSTs?







Node representations



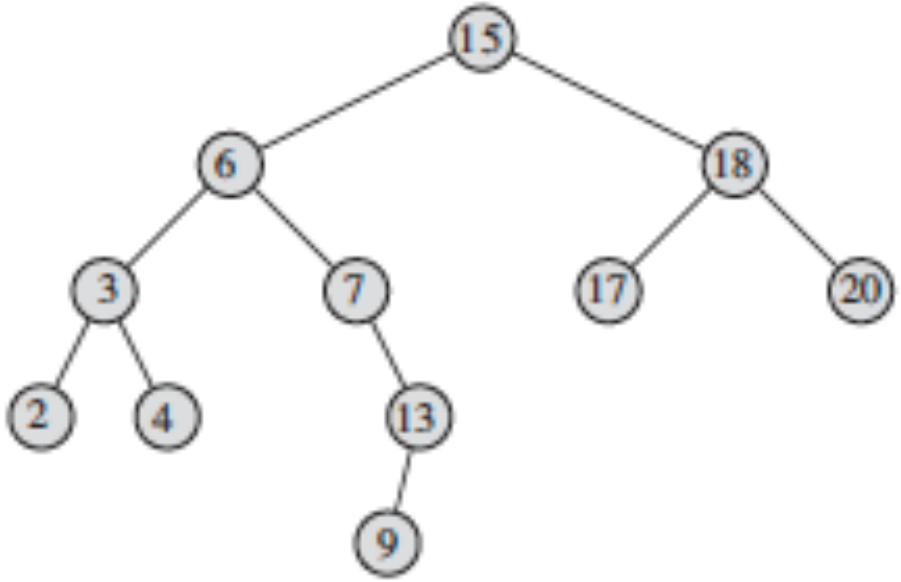
Lecture Plan

- Tree Data Structure
- Binary Search Tree
- Querying a Binary Search Tree
- Insertion and Deletion
- Summary

Search (lookup) Operation in BST

Runtime time: O(h)

```
TREE-SEARCH(t, k)
        if (t == null) or k == t.key
                return t
        if (k < t.key)
                return TREE-SEARCH(t.left, k)
        else    return TREE-SEARCH(t.right, k)
```

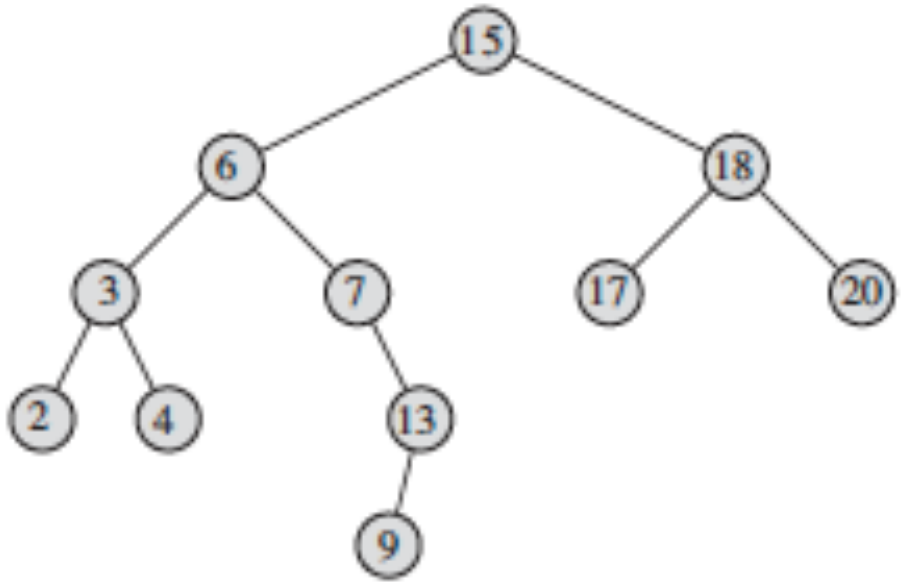


Exercise: Write an iterative version of TREE-SEARCH

Find Minimum Operation in BST

Runtime time: O(h)

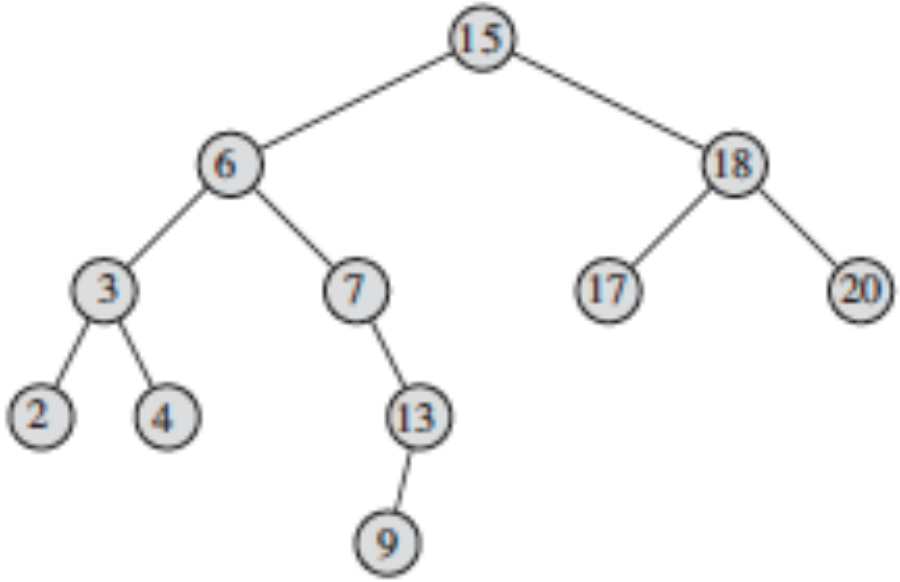
```
TREE-MINIMUM(t)
        while t.left != null
                t = t.left
        return t
```



Find Maximum Operation in BST

Runtime time: O(h)

```
TREE-MAXIMUM(t)
        while t.right != null
                t = t.right
        return t
```

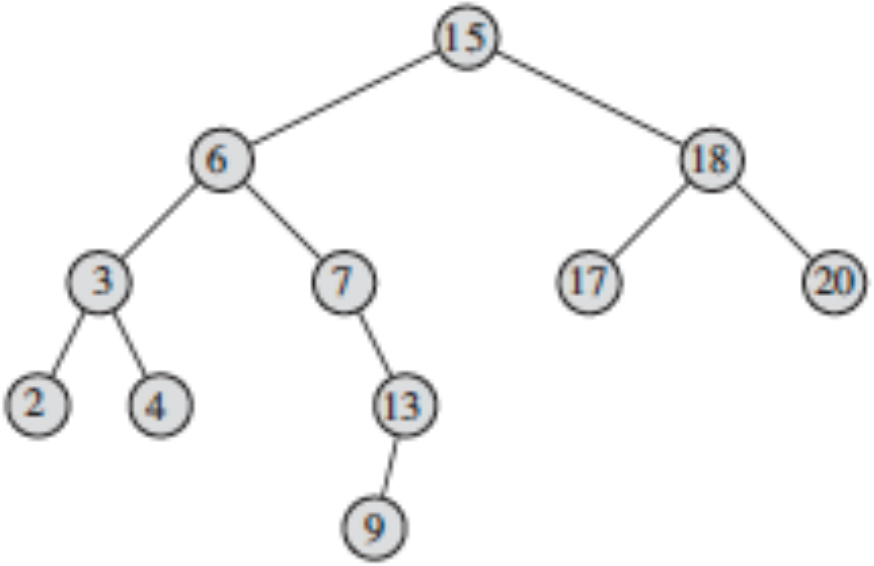


Find Successor Operation in BST

Runtime time: O(h)

```
TREE-SUCCESSOR(t)
        if t.right != null
                return TREE-MINIMUM(t.right)
        y = t.parent
        while y != null and t == y.right
                t = y
                y = y.p
        return y
```

Successor of a node ***t*** is the node with the smallest key greater than ***t***.key



Exercise: Write the TREE-PREDECESSOR procedure

Predecessor of a node ***t*** is the node with the largest key smaller than ***t***.key

Exercise

Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

Lecture Plan

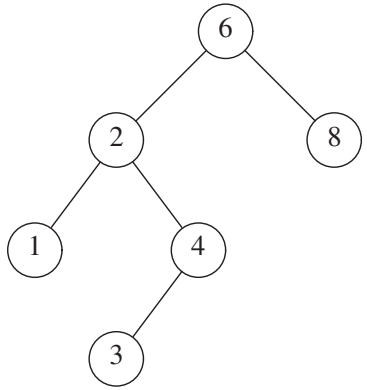
- Tree Data Structure
- Binary Search Tree
- Querying a Binary Search Tree
- Insertion and Deletion
- Summary

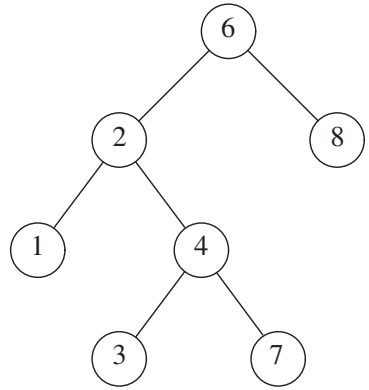
Insert Operation in BST

Runtime time: O(h)

- Do TREE-SEARCH(t, z.key). If z.key found, nothing to do.
- Otherwise, insert z at the last spot on the path.

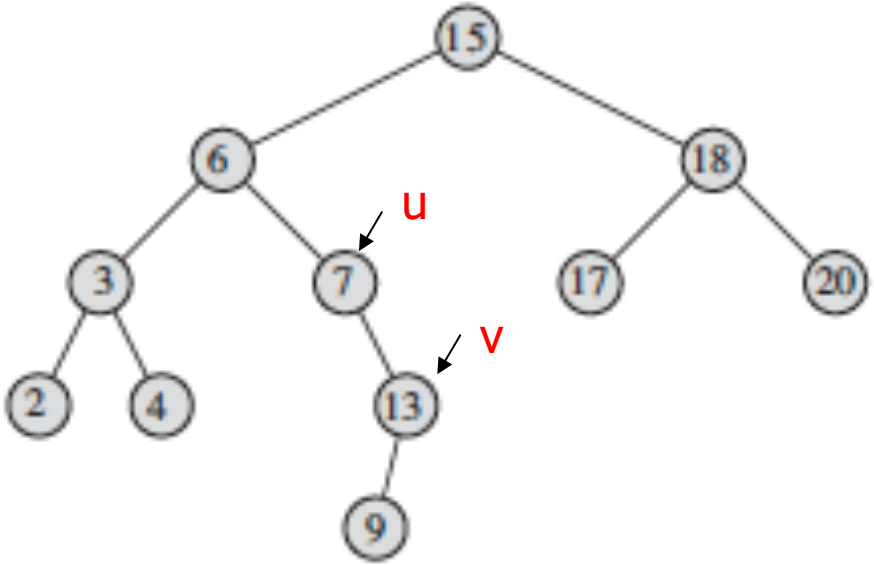
```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z          // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```





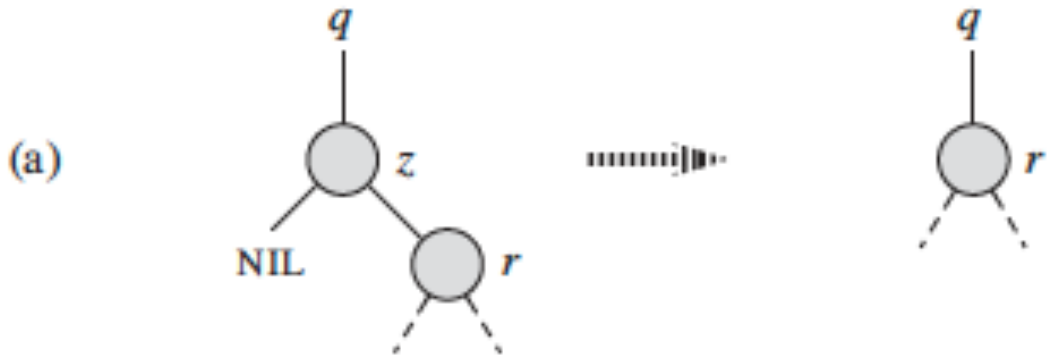
TRANSPLANT: Replace subtree rooted at u with the subtree rooted at v

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```

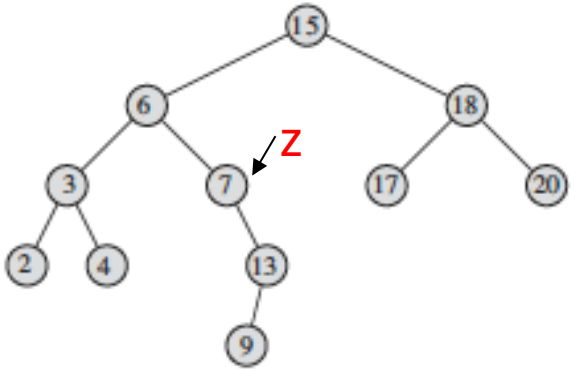


Delete Operation in BST

Case 1: if z has no left child, replace z by its right child

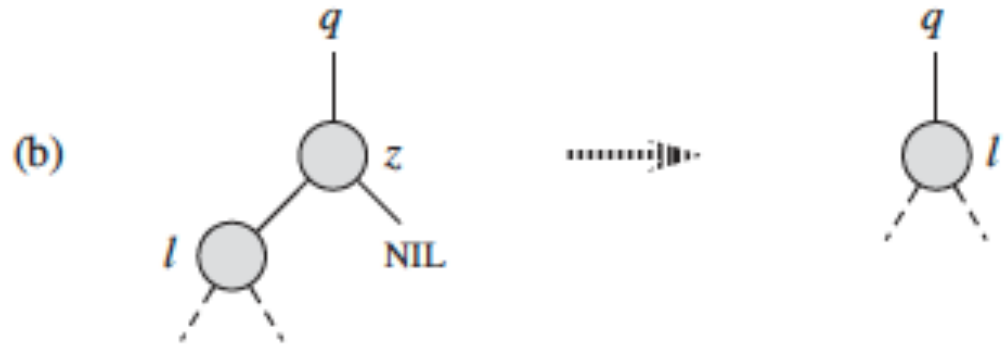



```
if z.left == null
        TRANSPLANT(T, z, z.right)
```

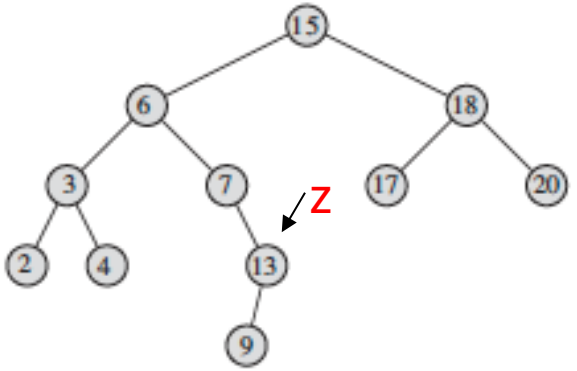


Delete Operation in BST

Case 2: if z has just one child, which is its left child, replace z by its left child



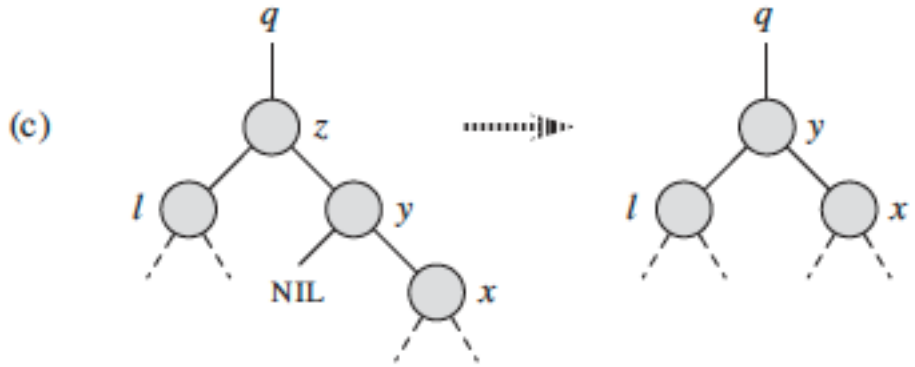

```
elseif z.right == null
        TRANSPLANT(T, z, z.left)
```



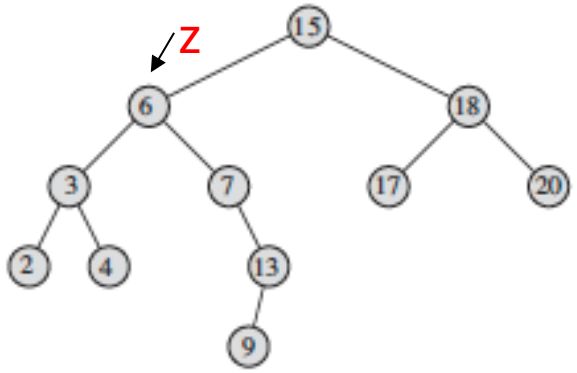
Delete Operation in BST

Case 3: z has both left and right child AND z's successor y is the right child of z,

→ we replace z by y




```
else
        y = TREE-MINIMUM(z.right)
        if y.parent == z
                TRANSPLANT(T, z, y)
                y.left = z.left
                y.left.parent = y
```

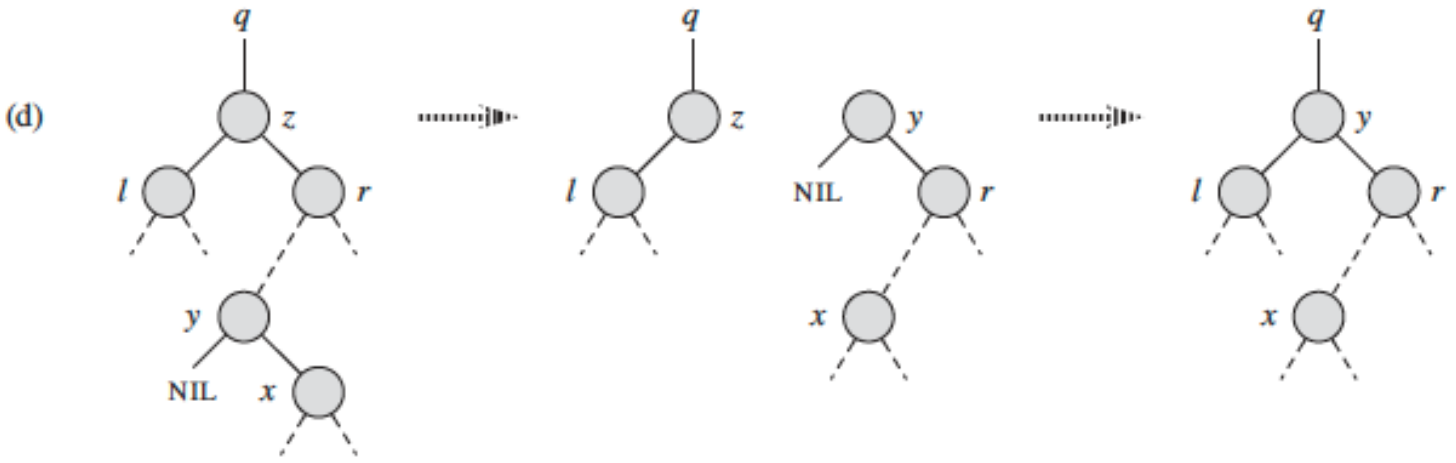


Delete Operation in BST

Runtime time: O(h)

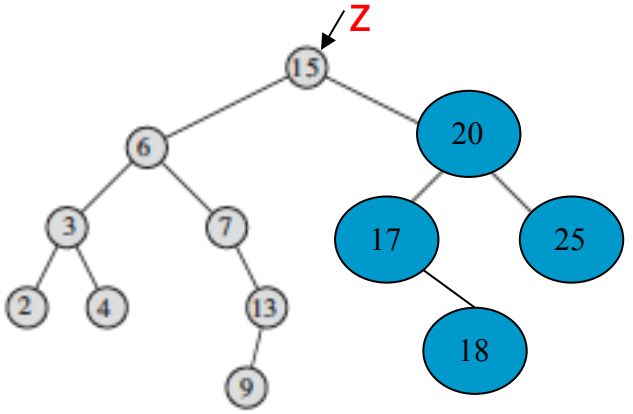
Case 4: z has both left and right child AND z's successor y is NOT the right child of z,

→ we first replace y by its right child, and then we replace z by y.



else

```
y = TREE-MINIMUM(z.right)
if y.parent != z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.parent = y
TRANSPLANT(T, z, y)
y.left = z.left
y.left.parent = y
```



Delete Operation in BST

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Lecture Plan

- Tree Data Structure
- Binary Search Tree
- Querying a Binary Search Tree
- Insertion and Deletion
- Summary

Summary

- search, findMin, findMax, findSuccessor, findPredecessor, insert, delete operations of a binary search tree of height h takes O(h)

- With appropriate balanced binary search trees, h is about log N

Using Binary Search Trees to implement Symbol Tables

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
sequential search (unordered linked list)	N	N	$N/2$	N	no
binary search (ordered array)	$\lg N$	N	$\lg N$	$N/2$	yes
binary tree search (BST)	N	N	$1.39 \lg N$	$1.39 \lg N$	yes